<알고리즘 Mix Tape 04>

8장_2 최소 신장 트리 (170605 to 170607)

우리가 어렸을 때 땅따먹기 게임을 해본 적이 있을 것이다. 벌레(혹은 지렁이었나...??)를 피해서 플레이어가 직사각형을 만들면 그림 조각이 나오게 되는데 그림 조각을 나오게 하기 위해 벌레를 피해서 큰 사각형을 만드는 경우가 있는데 벌레가 경계선을 넘어오면 플레이어도 죽는 모습을 볼 수 있다. 이처럼 가중 그래프에서도 수치가 가장 짧은 간선들을 선택해서 신장 트리를 만드는 과정이 바로 최소 신장 트리를 완성하는 원리이다. 최소 신장 트리의 알고리즘은 크게 2가지로 나뉘는데 크루스칼 알고리즘, 프림 알고리즘으로 나뉠 수 있다. 오늘은 두 알고리즘에 대해서 자세히 공부해보는 기회를 가져보도록 하자.

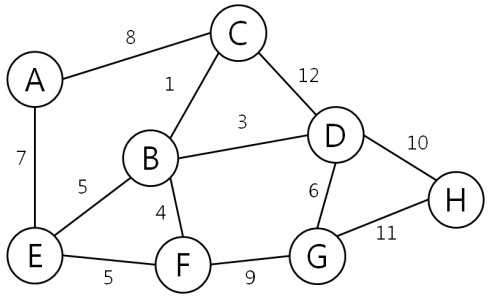
1. 최소 신장 트리(Minimal Spannin' Tree)

신장 트리는 8장-1에서 설명했는데 신장 트리가 되는 조건에 대해서 아래와 같이 살펴볼 수 있다.
-> 모든 정점에 대해서 인접해야 한다.
-> 간선의 수가 (정점-1)개여야 한다.
-> 트리 내부에 순환(사이클)이 있어서 안 된다.

신장 트리의 전제 조건에 대해서 숙지를 해야 이번에 최소 신장 트리를 공부하는데 있어서 어려움 없이 공부할 수 있다. 신장 트리는 일반 트리의 종류로 살펴볼 수 있다. 그렇지만 이번에 공부하는 최소 신장 트리는 도대체 어떤 뜻일까? 우리가 방금 DFS, BFS에 대해 공부했을 때 탐색을 하면서 남긴 발자취가 바로 각 탐색에 대한 신장 트리이듯이, 이번에는 가중치가 있는 그래프에 대해서 최소의 가중치로 구성된 신장 트리를 그리는 과정으로 살펴볼 수 있다. 말 앞에 최소를 통해서 무슨 뜻인지 감이 잡혔을 것이다.
여기서 강조하고 싶은 점은 최소 신장 트리를 그리는 과정이 절대로 경로를 구하는 과정으로 생각하면 큰 오산이다. 여기서는 최소의 가중치를 통해서 신장 트리를 완성하는 목표를 가졌는데 이를 통해 최소로 걸리는 경로를 구한다고 생각했다면 이는 다음에 배울 다익스트라 알고리즘을 참고해보길 바라면서 최소 신장 트리를 구하는 알고리즘인 Prim, Kruskal 알고리즘을 공부해보도록 하자.

2. Kruskal 알고리즘

우리가 최소 신장 트리를 가장 쉽게 추측을 할 수 있는 방법이 바로 크루스칼 알고리즘이다. 여기서 사행성 이야기를 꺼내면 안 되는데 로 자로 시작하는 복권을 해본 경험이 있는 사람은 OMR 카드에 컴퓨터 사인펜으로 아무 숫자들을 6개 찍어서 작성하거나 자동 선택으로 찍고 나중에 스포츠 뉴스 끝나고 심심할 때 맞춰본 경험이 있을 것이다. 우리가 당첨 여부를 쉽게 파악하기 위해서는 당첨 번호를 큰 순서대로 정렬을 해서 확인을 하는 방법이 있다. 자신이 찍은 번호가 당첨 번호에 포함되어 있으면 속으로 아싸를 하게 되는데 크루스칼 알고리즘은 정점과 정점 사이의 가중치를 큰 순서대로 정렬을 해서 작은 가중치부터 시작을 해서 신장 트리를 만들어가는 과정으로 보면 쉽다. 크루스칼 알고리즘을 이해하기 위하여 오른쪽 위와 같은 가중치 그래프를 준비하였다.



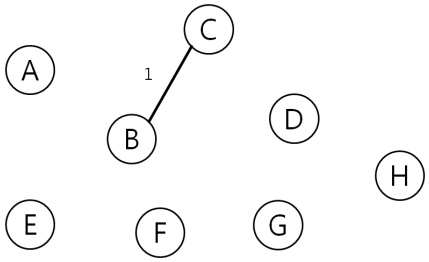
정점이 총 8개가 있기 때문에 지금 그래프에서 유지되어야 하는 간선의 수는 신장 트리의 조건에 맞추기 위해 7개를 채우는 것이 목적이다. 일단 간선들에 대해서 큰 순서대로 정렬을 하면 아래와 같이 나타낼 수 있다.

(B, C)	1	(A, E)	7
(B, D)	3	(A, C)	8
(B, F)	4	(F, G)	9
(B, E)	5	(D, H)	10
(E, F)	5	(G, H)	11
(D, G)	6	(C, D)	12

여기서 같은 가중치의 간선인 경우에는 시발점의 깊이가 낮은 것부터(깊이 우선 탐색에서 간선의 알파벳이 빠른 순서) 리스트에 저장을 하도록 한다. 그럼 이제부터 Kruskal 알고리즘이 어떻게 돌아가는지에 대해서 과정을 거쳐보도록 하자.

2-1) Kruskal 알고리즘의 과정

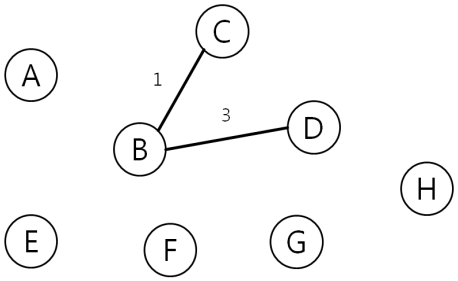
1단계) 가중치가 가장 작은 (B, C)를 그리자.



간선 (B, C)에 대해서 최소의 가중치인 1를 그리면 신장 트리의 조건을 지키는 범위 이내에서 쉽게 그릴 수 있다. 그럼 현재 신장 트리에 속하는 간선들의 집합의 요소에는 (B, C)가 들어가게 된다.

(B, C)	OK	(A, E)	7
(B, D)	3	(A, C)	8
(B, F)	4	(F, G)	9
(B, E)	5	(D, H)	10
(E, F)	5	(G, H)	11
(D, G)	6	(C, D)	12

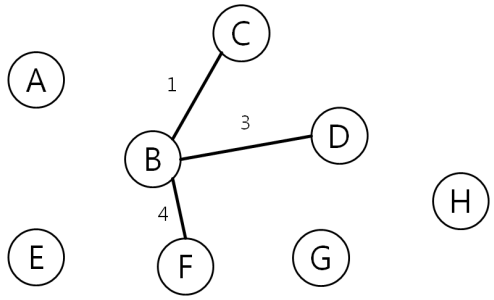
2단계) (B, D)를 그리자.



간선 (B, D)를 그리게 되어도 트리의 조건을 지키는 범위 이내에서 그리는데 이상이 없게 된다. 비록 현재 간선의 수가 부족해서 신장 트리의 전제 조건을 지키지 않았다고 생각을 하지만 아직까지 진행형이니 채우면서 완성을 한다고 생각하면 되겠다. 그럼 신장 트리의 간선 집합에는 (B, D), (B, C) 2개가 포함되어 있다.

(B, C)	OK	(A, E)	7
(B, D)	OK	(A, C)	8
(B, F)	4	(F, G)	9
(B, E)	5	(D, H)	10
(E, F)	5	(G, H)	11
(D, G)	6	(C, D)	12

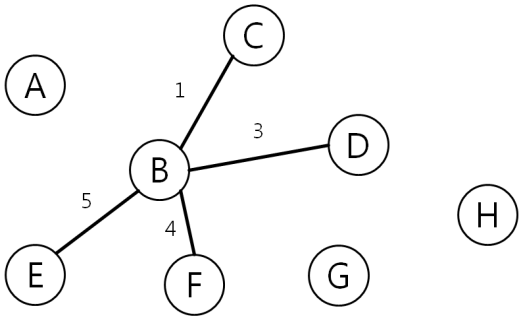
3단계) (B, F)를 그리자.



간선 (B, F)를 그리게 되어도 여태까지 트리의 조건을 위배하지 않았기 때문에 이상 없이 그릴 수 있다. 신장 트리를 이루는 간선의 집합으로 (B, C), (B, D), (B, F) 3개가 완성되었다.

(B, C)	OK	(A, E)	7
(B, D)	OK	(A, C)	8
(B, F)	OK	(F, G)	9
(B, E)	5	(D, H)	10
(E, F)	5	(G, H)	11
(D, G)	6	(C, D)	12

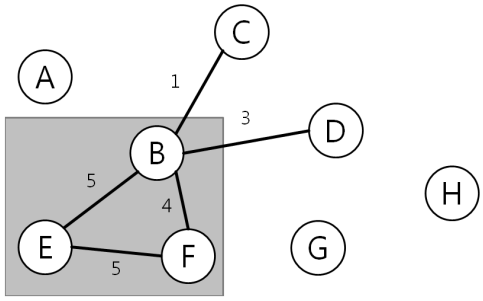
4단계) (B, E)를 그리자.



간선 (B, E)를 그리게 되어도 트리의 조건에 대해 만족하기 때문에 아직까지 이상이 없다. 그러나 문제가 되는 것은 다음 간선인 (E, F)를 그릴 때부터이다. 일단 신장 트리를 이루는 간선들의 집합에는 (B, C), (B, D), (B, F), (B, E) 로 4개이다.

(B, C)	OK	(A, E)	7
(B, D)	OK	(A, C)	8
(B, F)	OK	(F, G)	9
(B, E)	OK	(D, H)	10
(E, F)	5	(G, H)	11
(D, G)	6	(C, D)	12

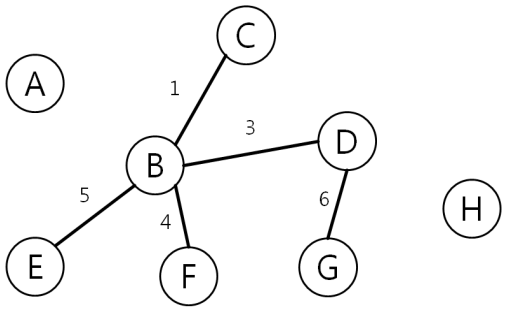
5단계_1) (E, F)를 그리게 되면...?



간선 (E, F)를 그리게 되면 정점 B, E, F 끼리 삼각형을 그리게 되어 순환(사이클)이 되어 부렸기 때문에 트리의 전제 조건에 대해 위배를 하게 된다. 그러면 (E, F)는 신장 트리를 이루는 간선의 집합에는 포함시키면 안 된다.

(B, C)	OK	(A, E)	7
(B, D)	OK	(A, C)	8
(B, F)	OK	(F, G)	9
(B, E)	OK	(D, H)	10
~~(E, F)~~	~~FAILED~~	(G, H)	11
(D, G)	6	(C, D)	12

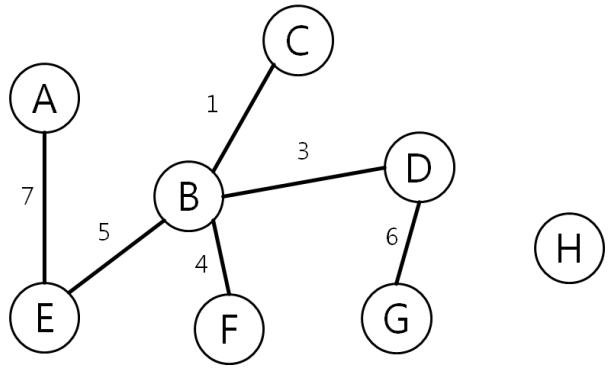
5단계_2) (D, G)를 그리자.



간선 (E, F)를 씹고 간선 (D, G)를 그리게 되면 아까처럼 순환(사이클)이 안 나왔기 때문에 트리의 전제 조건에 위배되지 않는 조건으로 신장 트리를 이루어 갈 수 있다. 신장 트리를 이루는 간선의 집합으로는 (B, C), (B, D), (B, F), (B, E), (D, G) 5개로 이루어진다.

(B, C)	OK	(A, E)	7
(B, D)	OK	(A, C)	8
(B, F)	OK	(F, G)	9
(B, E)	OK	(D, H)	10
~~(E, F)~~	~~FAILED~~	(G, H)	11
(D, G)	OK	(C, D)	12

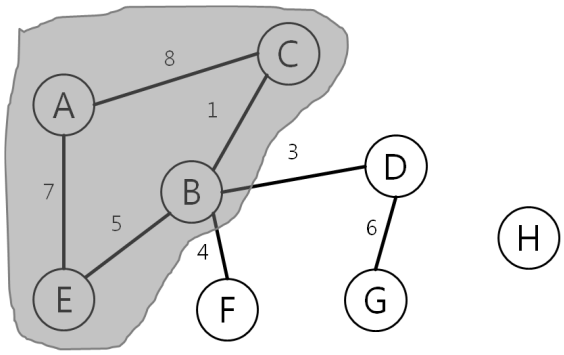
6단계) (A, E)를 그리자.



간선 (A, E)를 그리게 되어도 트리를 이루는데 있어서 문제가 없기 때문에 신장 트리를 이루어 갈 수 있다. 신장 트리를 이루는 간선의 집합으로서는 (B, C), (B, D), (B, F), (B, E), (D, G), (A, E) 총 6개가 된다. 현재 정점의 수는 8개이기 때문에 간선의 수는 7개로 마무리 때려야 한다는 사실을 알고 넘어가야겠다.

(B, C)	OK	(A, E)	OK
(B, D)	OK	(A, C)	8
(B, F)	OK	(F, G)	9
(B, E)	OK	(D, H)	10
~~(E, F)~~	~~FAILED~~	(G, H)	11
(D, G)	OK	(C, D)	12

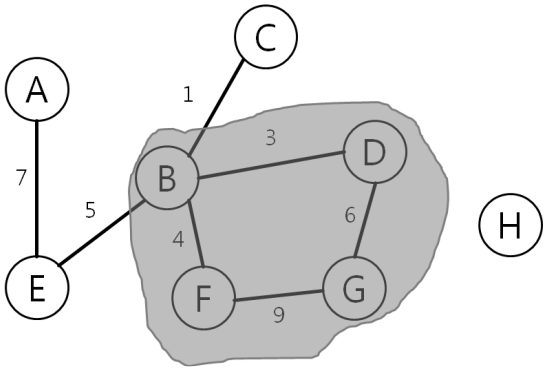
7단계_1) (A, C)를 그리게 되면...



그렇다. 정점 A, B, C, E끼리 이상한 사각형 모양을 만들게 되어 순환(사이클)을 이루게 되었기 때문에 트리의 조건에 위배된다. 간선 (A, C)는 신장 트리의 간선 집합에 포함시키면 안 된다.

(B, C)	OK	(A, E)	OK
(B, D)	OK	~~(A, C)~~	~~FAILED~~
(B, F)	OK	(F, G)	9
(B, E)	OK	(D, H)	10
~~(E, F)~~	~~FAILED~~	(G, H)	11
(D, G)	OK	(C, D)	12

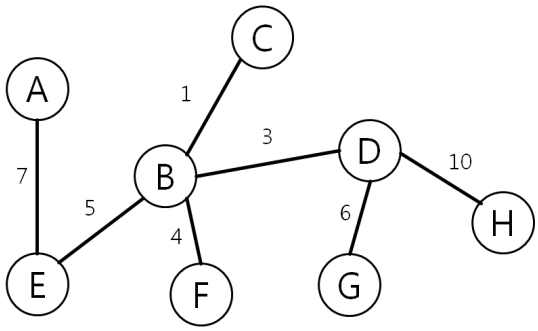
7단계_2) (F, G)를 그리게 되면...



아까와 마찬가지로 정점 B, D, F, G끼리 모여서 사다리꼴 같은 모양으로 순환(사이클)로 이루게 되어 트리의 조건에 위배된다. 간선 (F, G)는 신장 트리의 간선 집합에 포함시키면 안 된다.

(B, C)	OK	(A, E)	OK
(B, D)	OK	~~(A, C)~~	~~FAILED~~
(B, F)	OK	~~(F, G)~~	~~FAILED~~
(B, E)	OK	(D, H)	10
~~(E, F)~~	~~FAILED~~	(G, H)	11
(D, G)	OK	(C, D)	12

7단계_3) (D, H)를 그리자.



간선 (D, H)를 그리게 되면 최종적으로 모든 정점에 대해서 간선이 연결이 되었으며, 트리의 조건에 위배되지 않은 범위 이내에서 최소 신장 트리가 완성된다. 최종적으로 간선의 집합으로서는 (B, C), (B, D), (B, F), (B, E), (D, G), (A, E), (D, H)로서 정점 8개 중에서 간선 7개가 나오게 되니 올바르게 그린 정답이 되겠다.

(B, C)	OK	(A, E)	OK
(B, D)	OK	~~(A, C)~~	~~FAILED~~
(B, F)	OK	~~(F, G)~~	~~FAILED~~
(B, E)	OK	(D, H)	OK
~~(E, F)~~	~~FAILED~~	(G, H)	볼 필요가 없게 된다.
(D, G)	OK	(C, D)	

Kruskal 알고리즘의 과정을 보니깐 크게 어렵게 느끼진 않을 것이다. 그렇지만 실제로 Kruskal 알고리즘을 알고리즘에 접목하기 위해서는 신장 트리를 이루는 간선들의 집합으로 따져 나가야되기 때문에 집합(Set) 관련 알고리즘이 필요하다. Kruskal 알고리즘은 어떻게 구성 되어있는지 아래를 살펴보도록 하자.

```
Kruskal (G, r) {
  T←Φ ; // T 신장트리에 빈 집합을 대입한다. // 1
  단 하나의 정점만으로 이루어진 n 개의 집합을 초기화한다; // 2
  모든 간선을 가중치가 작은 순으로 정렬한다; // 3
    while (T 의 간선수 < n - 1) {
      최소비용 간선 (u, v)를 제거한다; // 4
      if 정점 u 와 정점 v 가 서로 다른 집합에 속하면
      {
         두 집합을 하나로 합친다;
         T ← T ∪ { (u, v ) }; // 5
       }
     }
}
```

크루스칼 알고리즘은 우리가 실제로 그래프를 지우면서 작성을 하면 인간의 본능을 이용해서 정점들 간의 순환(사이클)이 발생했는지에 대해서 판단을 하기 때문에 쉽게 생각하겠지만 실제로 컴퓨터는 0이랑 1밖에 모르는 똥개이기 때문에 우리가 컴퓨터의 지능을 맞춰서 작성을 한 코드가 위처럼 작성이 되겠다.

1 : 여기서 신장 트리에는 맨 처음에는 아무 것도 들어있지 않아서 공집합으로 초기화를 해준다.

2 : 여기서 하나의 정점만으로 이루어진 n개의 집합을 초기화한다는 뜻은 각 정점의 개수가 n개인 경우에 정점 별로 인접한 다른 정점에 대해 기록을 하는 동시에 가중치까지 기재하는 원리로 생각하면 되겠다.

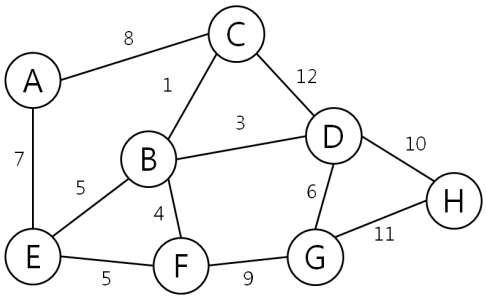
3 : 여기서 모든 가중치가 작은 순으로 정렬해야 하는 것은 크루스칼 알고리즘의 일반적인 원칙이다. 그래서

작은 값들부터 정렬을 해서 큰 값까지 살피면서 순환(사이클)이 존재하는가 안 하는가에 대해 따지면 된다.
4 : 여기서 T의 간선 수는 정점의 n-1보다 작을 때까지라는 뜻은 본래 신장 트리의 간선의 수가 정점-1이기 때문에 이 철칙을 지키기 위하여 While 문으로 쓰면서 이용하는 방법이라 생각하면 되겠다. 여기서 각 정점들에 대해 오름차순으로 지워 나가면 된다.
5 : 여기서 T는 신장 트리를 완성하기 위한 간선의 집합으로 생각하면 된다. 만일 u, v가 서로 같은 집합에 속한다는 뜻은 쉽게 생각하면 (u, v1) (v1, v2) (v2 v)...처럼 (v, u)가 추이적으로 나온 결과가 되어 같은 집합 내에서도 정점만 바뀌어도 결국 순환(사이클)을 형성한다는 뜻으로 생각하면 되겠다. 그래서 이를 만족하지 않아야 순환(사이클)이 안 나오기 때문에 이를 만족하지 않은 간선들로 이루어서 신장 트리를 완성하면 되겠다.

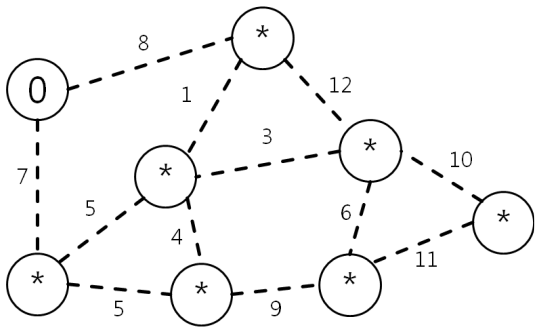
크루스칼 알고리즘은 본래 이렇게 어려운 이론들을 넣어야 하기 때문에 실제로 공부하기에는 벅찰 것이다. 그래서 이번에는 프림 알고리즘을 공부해보도록 하자.

3. Prim 알고리즘

옛날에 우리가 커피를 타 마실 때 프림을 타 마시는 경우가 있다. 요즘은 식물성 기름이 건강에 안 좋아서 카제인 나트륨으로 대체를 하지만 뭐... 그것 보다 시작 정점에서 주변에 있는 간선들의 가중치를 비교하면서 최소 신장 트리를 만드는 과정으로 생각하면 되겠다. 그렇지만 프림 알고리즘은 다음에 배우는 다익스트라 알고리즘에서도 우려먹을 정도로 중요한 원리이니 프림 알고리즘의 원리를 익혀두도록 하자. 아까와 같은 그래프를 통해서 Prim 알고리즘이 어떻게 돌아가는지에 대해 공부해보도록 하자.

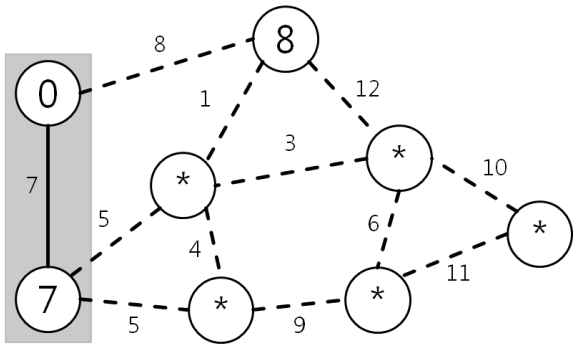


1단계) 모든 정점을 무한대(여기서는 *로 대체할게요...)로 수치를 바꾼 후에 시작점인 A는 0으로 바꿔두록 한다.



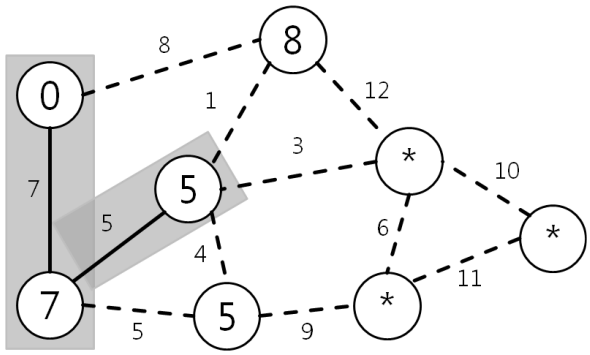
더더욱 재미있는 것은 시작 정점 이외에 타 정점들은 다익스트라 알고리즘에서도 우려먹는 사실을 짚을 수 있다.

2단계) A 정점 주변에 있는 최소 간선을 찾아내자



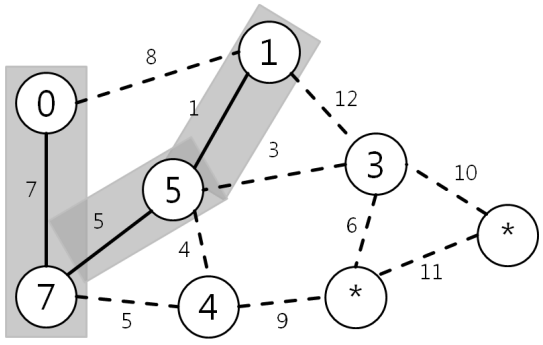
A는 현재 C, E랑 연결이 되어 있는데 이 둘 중에서 가장 낮은 가중치가 7인 E랑 연결을 해줘야 되겠다. 이제 A와 E(7)가 연결된 간선들 중에서 최소를 따져보도록 하자.

3단계) 같은 가중치일지라도 깊이가 낮은 녀석부터 채워주자



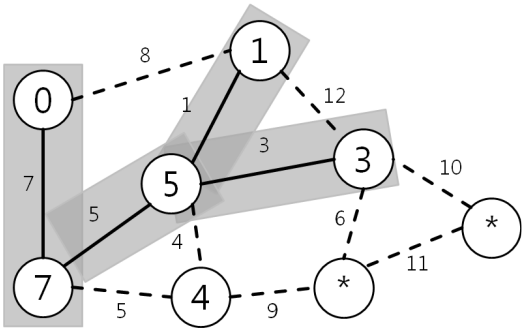
정점 A, E로 이루어진 간선에 인접하는 정점들을 살펴보면 B(5), C(8), F(5)로 나뉘어 볼 수 있다. 그렇지만 여기서 가중치가 제일 낮은 간선은 BE랑 FE로 살펴볼 수 있다. 그렇지만 Prim 알고리즘은 깊이가 낮은 정점들에 대해서 우선적으로 살펴본다고 생각하면 되겠다. 그러면 A랑 E, E랑 B로 연결할 수 있음을 인지할 수 있을 것이다.

4단계) A, E, B끼리 인접한 정점들 중에서 최솟값을 찾아내자



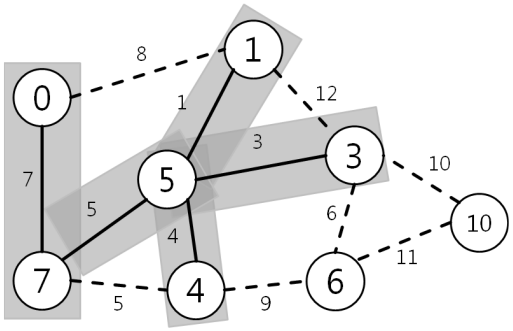
그럼 정점 A, B, E로 이루어진 간선에 인접하는 정점들을 살펴보면 C(8->1로 바뀜. 최소를 저장하니깐...), D(3), F(5->4로 바뀜. 최소를 저장하니깐...)로 바뀐 이후에는 여기서 가중치가 최소인 1인 C랑 연결을 해준다. 그럼 정점은 A, E, B, C로 이루어지게 된다.

5단계) A, E, B, C 중에서 인접한 정점들 중에 최솟값을 찾아내자 -> Next Page ->



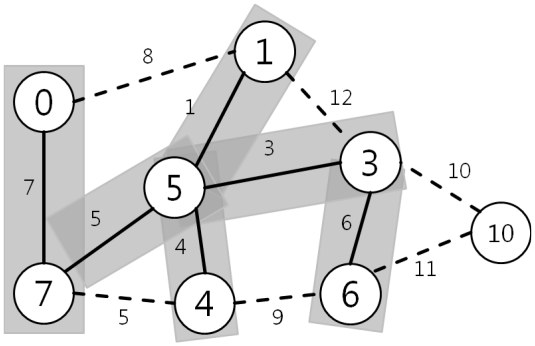
우선 정점 A, E, B, C 중에서 인접한 정점 중에 최소값은 D로서 3인 간선과 연결이 된다. 결국 신장 트리 집합에 모인 정점들은 A, E, B, C, D가 되겠다.

6단계) A, E, B, C, D 정점들 중에서 최솟값 파기

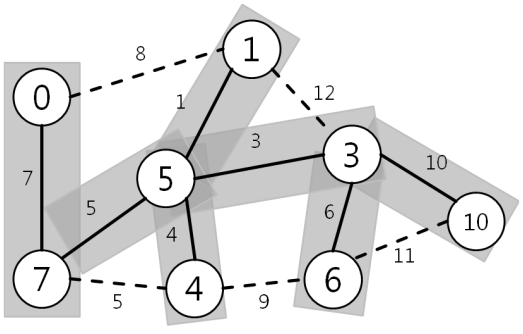


이제 F, G, H에 대해서도 최솟값을 결정할 때가 슬슬 다가오고 있다. 우선 F의 값은 4로 바뀌게 되고, G는 6으로 바뀌게 된다. 또한 H는 10으로 바뀌게 되어 최종적으로 최솟값은 F랑 연결함으로서 다음 단계로 넘어간다.

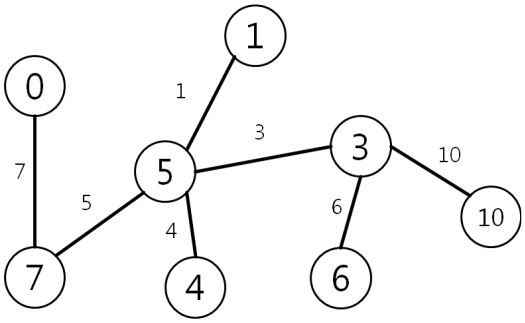
7단계) G, H 정점들에 대해서 완성하기



F에서 G로 이동하는 가중치, D에서 G로 이동하는 가중치에서 가장 최소인 것은 D에서 G로 다이렉트로 이동하는 가중치이기 때문에 위처럼 간선을 채울 수 있다.



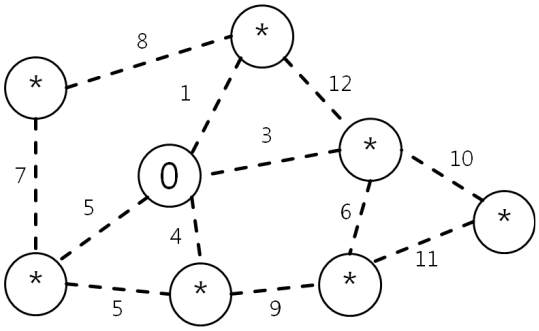
이제 최종적으로 H로 향하는 간선들 중에서 최소값인 녀석을 골라내야 하는데 D에서 H로 이동하는 간선을 통해서 최소 가중치를 선택할 수 있음으로서 이렇게 최소 신장 트리가 완성이 된다.



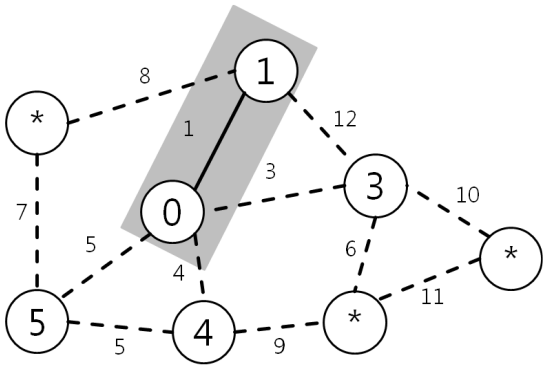
Prim 알고리즘은 크루스칼 알고리즘과 달리 시작하는 정점을 임의로 고를 수 있다. 그렇지만 최소 신장 트리의 결과는 위와 다를 바가 전혀 없다.(다만 자바의 결과 창에서는 출력 순서가 다르겠죠...??) 이번에는 Prim 알고리즘이 어떻게 돌아가는지에 대해 공부해보자.

```
public static void primAlgorighm(HashMap<String, 간선[]> 그래프, String 시작정점) {
    HashSet<String> 방문한정점 = new HashSet<>();
    방문한정점.add(시작정점); // 1
    while (방문한정점.size() < 그래프.size()) {
     간선 최소간선 = // 2
        new 간선(null, Integer.MAX_VALUE, null);
     for (String 정점 : 방문한정점)
        for (간선 간선 : 그래프.get(정점)) // 2
          if    (방문한정점.contains(간선.정점2)    == false && 간선.가중치 < 최소간선.가중치) // 3
            최소간선 = 간선;
        방문한정점.add(최소간선.정점2); // 4
        System.out.printf("%s %d %s\n", 최소간선.정점1, 최소간선.가중치, 최소간선.정점2); // 4
    }
}
```

1 : 일단 prim 알고리즘은 임의의 정점들 중에서 하나를 잡아서 시작을 하는 원리로 생각을 하면 되겠다. 여기서 방문한 정점에 대해서 추가를 해주고 난 후에
2 : 최소간선 이란 변수를 통해서 방문했던 정점들을 기준으로 인접한 정점으로 연결된 간선의 최소 가중치를 이용해서 찾을 필요가 있다. Java에서는 int의 가장 최댓값인 MAX_VALUE를 통해 가상의 무한대로 생각을 할 수 있는데 원래 Prim 알고리즘에서도 시작하는 정점을 빼고 나머지 정점들에 대해서는 무한대로 생각을 하고 시작해야 한다.
3 : 방문한 정점들이 포함되었는가에 대해서는 contains 메소드를 통해서 타 정점에 대해 방문을 했는지 확인을 하는데 이용이 된다. 또한 현재 정점에 대해 인접한 정점들에 대한 간선을 탐색하면서 가중치를 비교를 때려준다. 잠깐 그림으로 어떠한 원리로 돌아가는지에 대해 알아보고 넘어가자.



이번에는 A 정점에서 시작을 하면 이해할 수 없으니 B 정점에서 시작을 해보도록 하자. 그러면 정점 B는 이미 방문을 했으니 B는 방문 처리가 되었고, B에 인접한 정점들인 C, D, E, F별로 인접한 정점들의 거리를 조사 한다.



그러면 정점 B랑 인접한 정점들 중에서 C(1), D(3), E(5), F(4) 중에서 가장 가중치가 낮은 간선이 C랑 연결된 간선이다. 이젠 정점 B와 C와 인접한 정점들을 순서대로 비교해보면서 최소 신장 트리를 만들어간다고 생각하면 되겠다. 이 작업을 3번에서 한다고 보면 되겠다.
4 : 최소간선에서 정점2란 변수를 잡아오는 이유가 현재 정점을 정점1로 가정을 하고 현재 정점과 인접한 정점에 대해서는 정점2로 생각을 할 수 있기 때문에 정점2를 방문 처리를 하고 난 후에 최소 간선에 대해서 출력을 하는 것으로 종결하는 것으로 볼 수 있겠다.

4. Prim 알고리즘의 시간 복잡도와 개선

아까와 같이 알아본 프림 알고리즘의 시간 복잡도는 어떻게 되는지에 대해서 생각해본 적은 없을 것이다. 그렇지만 본래 Prim 알고리즘의 시간 복잡도는 O(E log V)(V는 정점의 수, E는 간선의 수로 간주하겠음.)가 나올 정도로 줄일 수 있는데 아까와 같은 알고리즘으로 시간 복잡도를 돌리는 과정을 계산해보자. 그 코드를 복붙하면...

```
public static void primAlgorighm(HashMap<String, 간선[]> 그래프, String 시작정점) {
    HashSet<String> 방문한정점 = new HashSet<>();
    방문한정점.add(시작정점); // 0
    while (방문한정점.size() < 그래프.size()) { // 1
     간선 최소간선 =
        new 간선(null, Integer.MAX_VALUE, null);
     for (String 정점 : 방문한정점) // 2
        for (간선 간선 : 그래프.get(정점)) // 3
          if (방문한정점.contains(간선.정점2) == false && 간선.가중치 < 최소간선.가중치)
            최소간선 = 간선;
        방문한정점.add(최소간선.정점2); // 4
        System.out.printf("%s %d %s\n", 최소간선.정점1, 최소간선.가중치, 최소간선.정점2);
    }
}
```

0 : 해시 테이블, 해시 집합 등은 실상 시간 복잡도랑 크게 연관이 없다. 어자피 이 녀석들의 시간 복잡도는 O(1)이 나오기 때문이다.
1 : 여기서 방문한 정점의 수가 모두가 포함될 때까지 반복을 해야 하기 때문에 1번 반복문내에서 실행되는 횟수는 정점의 수인 V만큼 실행된다고 보면 되겠다.
2 : 방문한 정점들에 대해서 탐색을 하는 경우에는 어자피 정점들 중에서 탐색을 하는 것이니 V만큼 실행된다고 생각하면 되는데... 문제는 3번이다.
3 : 방문한 정점에 대해 인접한 정점과 연결된 간선에 대해 따져야 하는데 우리가 이용한 그래프 알고리즘에는 무향 가중치 그래프로서 간선의 정보가 가령 AC로 8인 경우에 CA이고 8인 간선의 정보가 또 들어있다는 것이다. 그래서 간선의 수의 2배인 만큼 반복문이 돌아간다고 생각을 하면 되겠다.
4 : 아시다시피 해시 테이블, 해시 집합에 대해서는 시간 복잡도에 크게 관여하지 않기 때문에 신경 쓰지 않고 넘어가면 되겠다.

그러면 1~4를 토대로 시간 복잡도를 계산해보면 어떠한 결과가 나올까? 여기서 변수에 대한 정의를 딱 잡고 가야 할 필요가 있는 점이 그래프에 존재하는 간선의 수는 E로 잡되, 그래프의 정점의 수는 V로 잡는다. 여기서 E는 V보다 작으면 안 된다는 점을 생각할 수 있다. 왜냐면 여기서는 정점의 수가 V인 것을 그대로 치고 모든 정점이 연결되어 있다고 가정을 한다면, 가중 그래프를 다루는데 정점의 수보다 작으면 결국에는 신장 트리가 나와 버리기 때문이다. 그래서 E랑 V를 별개로 볼 수밖에 없다.
O(V) × O(V) × O(E) => 1, 2, 3번을 토대로 곱해준다.
이를 정리하면 O(V^2+V×E)가 나오게 되는데 V^2보다 V×E가 더 크다는 점을 감수해야 한다. 그러면 결국에는 O(V×E)의 시간 복잡도가 나오게 된다.

그렇지만 시간 복잡도를 고려해본다면 이것보다 더 낮게 나올 수도 있을텐데 라는 생각도 해본 사람도 있을 것이다. 그래서 이번에 새로 올라온 자료를 토대로 참고해보도록 하자.

4-1) 간선의 저장 공간에 대해 바꿔보자

```
static Vertex[] createGraph() {
  Vertex[] V = new Vertex[] {new Vertex("A"), new Vertex("B"), new Vertex("C"), new Vertex("D"), new Vertex("E"), new Vertex("F"), new Vertex("G") };
  final int A = 0, B = 1, C = 2, D = 3, E = 4, F = 5, G = 6;
 V[A].edge = new Edge[] { new Edge(11, V[B]), new Edge(9, V[C]), new Edge(8, V[D]) };
 V[B].edge = new Edge[] { new Edge(11, V[A]), new Edge(13, V[C]), new Edge(8, V[E]) };
 V[C].edge = new Edge[] { new Edge(9, V[A]), new Edge(13, V[B]), new Edge(12, V[F]), new Edge(5, V[G]) };
 V[D].edge = new Edge[] { new Edge(8, V[A]), new Edge(10, V[G]) };
 V[E].edge = new Edge[] { new Edge(8, V[B]), new Edge(7, V[F]) };
 V[F].edge = new Edge[] { new Edge(12, V[C]), new Edge(7, V[E]) };
 V[G].edge = new Edge[] { new Edge(5, V[C]), new Edge(10, V[D]) };
 return V;
}
```

여기서 정점에 있는 edge라는 맴버는 아시다시피 인

접한 정점들에 대한 정점과 그 정점과 연결된 가중치에 대해 정리한 간선들의 배열로 보면 되겠다. 아까는 정점1, 가중치, 정점2 형식으로 저장이 되어 있어서 같은 정점에 정점1을 넣는 방식이 복잡해지고 그렇게 쓸모가 없기 때문에 이러한 방법대로 정리를 하는 방법이 존재한다.

```
static Vertex extractMin(Vertex[] V, HashSet<Vertex> visited, HashMap<Vertex, Integer> distance) {
  int min = Integer.MAX_VALUE;
  Vertex vertex = null;
  for (Vertex v : V) {
   if(visited.contains(v) == false && distance.get(v) < min) {
     vertex = v;
     min = distance.get(v);
    }
  }
  return vertex;
}
static void prim (Vertex[] V, Vertex start, HashMap<Vertex, Integer> distance, HashMap<Vertex, Vertex> previous) {
 HashSet<Vertex> visited = new HashSet<>();
 for(Vertex v : V)
    distance.put(v, Integer.MAX_VALUE);
 distance.put(start, 0);
 while (visited.size() < V.length) {
   Vertex u = extractMin(V, visited, distance);
 visited.add(u);
 for (Edge e : u.edge) {
   Vertex v = e.vertex;
  if (visited.contains(v) == false && distance.get(v) > e.weight) {
   distance.put(v, e.weight);
   previous.put(v, u);
   }
  }
 }
}
```

이 코드는 아까 우리가 살펴본 Prim 알고리즘과 별반 다를 바가 없는 코드로 보일 수도 있는데 여기에 HashMap이 2개가 추가된 것을 볼 수 있다. distance랑 previous 해시 맵이 추가되었는데 distance는 각 정점의 알파벳을 인덱스로 잡아서 각 인접한 정점에 있는 간선들 중 최소 신장 트리를 만족하는 최솟값들을 저장하기 위해 잡아뒀다. 그리고 previous는 방문 완료한 정점들 중에서 인접한 정점들로 향하는 각 간선들의 가중치가 낮은 정점을 저장해두되 최소 신장 트리를 만족하는 조건에 대해 저장을 해뒀다. 이러한 방안을 쓸지언정 시간 복잡도에는 애석하게도 변화가 없다. 특히 최솟값을 골라내는 부분에 대해서 살펴볼 수 있겠는데 굳이 저렇게 선형적으로 최솟값을 골라오는데 오히려 시간 낭비일 뿐이다. 그래서 최솟값을 골라오는 알고리즘에 대해서 변형을 줘보도록 하자.

4-2) Heap 자료구조를 접목시켜보자

```
static void prim(Vertex[] V, Vertex start, HashMap<Vertex, Vertex> previous) {
    VertexHeap heap = new VertexHeap(V);
    start.distance = 0;
    heap.heapifyUp(start.index);
    while (heap.size() > 0) {
      Vertex u = heap.extractMin();
      u.visited = true;
      for (Edge e : u.edge) {
        Vertex v = e.vertex;
        if (v.visited == false && v.distance > e.weight) {
          v.distance = e.weight;
          heap.heapifyUp(v.index);
          previous.put(v, u);
        }
      }
    }
}
```

여기서 정점에도 distance를 둬서 최솟값을 따지면서 Prim 알고리즘이 돌아가게끔 하는 원리로 접목을 시켰다고 볼 수 있는데 여기서 heap은 정점들 별로 간선의 가중치를 측정하여 최소인 값들에 대해서 저장을 하기 위해 만든 자료구조로 보면 되겠다. heap 자료구조를 이용해서 최솟값을 골라내면 아까 간선들의 정점 별로 선형 탐색을 하는 것과 달리 힙을 이용해 최솟값을 골라냄으로서 시간 복잡도가 O(E log V) 로 줄어들게 된다.